Collections

Rémi Forax

Historique

Java 1.0: pas de collection (1995)
- Vector, Stack, Hashtable

Java 1.2: API des collections (1998)
- List, Set, Map et ArrayList, HashSet, HashMap

Java 1.5/1.6: Generics + queue + collections concurrentes (2004-2007)
- Queue/Deque, CopyOnWriteArrayList, ConcurrentHashMap

Java 1.8: Lambdas (2014)
- Collection.removeIf(), List.sort(), Map.computeIfAbsent()

Java 11: Collection non modifiable (2018)
- List.of/copyOf(), Set.of/copyOf, Map.of/copy()

Design de l’API des collections (partie 1)

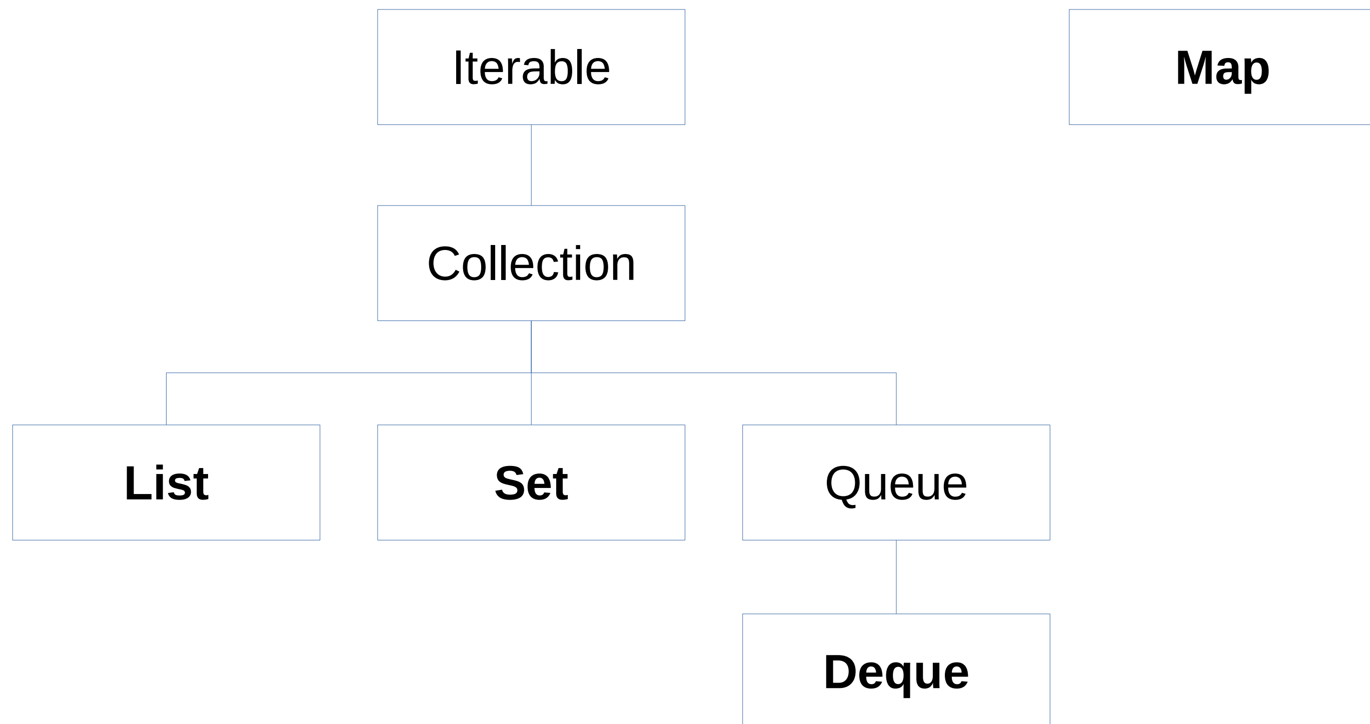
Design de l’API des collections

- Plusieurs interfaces / implantations
- Vue
- Mutabilité
- Nullabilité
- Ordre
- Concurrence
- Iterateur et itération

Interfaces

L’API définie les interfaces

- List, Set, Deque et Map



Interfaces et Implantations

L'API définie les interfaces

List, Set, Deque et Map

Deux sortes d'implantations

- Les implantations nommées
  - ArrayList, HashSet, ArrayDeque, HashMap
- Les implantations anonymes
  - List.of(), Arrays.asList(), List.subList(), Map.copyOf()

Contrat sur les éléments/clés

Les interfaces sont paramétrés

– List<E>, Set<E>, Deque<E> et Map<K,V>

Les éléments (E) ou les clés (K) doivent implanter les méthodes

– equals(Object): un élements est égal à n'importe quel objet
– hashCode(): un entier résumant les champs de l'objet
– toString(): un affichage

Exemple si on oublie hashCode()

```
class Person {
  private int age;

  public Person(int age) { this.age = age; }

  public boolean equals(Object o) {
    return o instanceof Person p && age == p.age;
  }
}

var person = new Person(32);
var set = Set.of(person);
System.out.println(set.contains(new Person(32)));  // false
```

Contrat sur les éléments/clés (2)

Les éléménts ne doivent **pas** être **modifiés** après insertion

```
class Person {
  ...
  public int hashCode() { return age; }
}

var person = new Person(32);
var set = Set.of(person);
person.age = 23;  // mutation ahhh
System.out.println(set.contains(person));  // false
```

Interface vs Implantation

Doit-on la manipuler une collection par son interface ou par son implantation ?

List<String> list = **new** ArrayList<>();

ou

ArrayList<String> list = **new** ArrayList<>()

Interface vs Implantation (2)

Quels sont les problèmes ?

- Si une méthode prend en paramètre une implantation, on ne peut pas en envoyer une autre

```
public void m(ArrayList<String> list) { … }
```

- Une opération d’une interface a une complexité differente suivant l’implantation

```
public void m(Set<String> list, String value) {
  set.contains(value);
    // HashSet ou Set.of(): compexité O(1)
    // TreeSet            : complexité O(ln n)
}
```

Encapsulation

Si la collection fait partie de **l’API**

- On utilise l’**interface** (paramètres, type de retour des méthodes publiques)
- sinon on utilise l’implantation (champ privé, variable locale)

```
public class Library {
  private final ArrayList<Book> books = new ArrayList<>();

  public List<Book> getAllBooksPlus(Book b) {
    var result = new ArrayList<>(books);  // var aide ici !
    // ArrayList<Book> result = new ArrayList<>(books);
    result.add(book);
    return result;
  }
}
```

Interface et Mutation

L’API utilise une même interface pour les implantations modifiables et non modifiables

- Cela évite d’avoir trop d’interfaces

donc les méthodes qui modifient la collection sont *optionnelles*

- add/remove/set/replace/clear etc

  elles peuvent lever l’exception UnsupportedOperationException

Interface et Mutation (2)

On ne mute pas une collection que l'on a pas soit même créée

```
static void removeLastIfEmpty(List<String> list) {
  var last = list.get(list.size() - 1);
  if (last.isEmpty()) {
    list.remove(list.size() - 1);  // ahhhh
  }
}
```

```
static List<String> removeLastIfEmpty(List<String> list) {
  var last = list.get(list.size() - 1);
  if (last.isEmpty()) {
    return list.stream().limit(list.size() - 1).toList();  // ok !
  }
  return list;
}
```

Modification *structurelle*

L'API définie deux sortes de modifications

- Les modifications qui change la structure (size change)

  ex: List.add, Set.remove

- Les modifications non-structurelles
  - ex: List.set ou Map.replace

ex: List.asList() permet les modifications non-structurelles

```
var list := List.asList(1, 2, 3);
list.add(4);       // UnsupportedOperationException
list.set(0, 42);  // ok !
```

E/K or Object ?

Les méthodes de recherche contains(), remove() ou Map.get() prennent un Object en paramètre pas un E/K

- Car on peut rechercher un objet avec un type différent que celui des éléments / clés

```
class A implements I, J { }

A a = new A()
I I = a;  // sous-typage
J j = a;  // sous-typage
List<I> list = List.of(i);
list.contains(j)   // contains doit être typé Object
```

Valeur de retour des méthodes

Les méthodes qui font des effets de bords (qui modifie une Collection/Map) renvoie un booléen pas void

- true si la collection/map a été modifiée

Permet de savoir l'opération a été effectuée

- Set.add(E) renvoie false si l'élement déjà présent
- Map.replace(K,V,V) renvoie vrai si la valeur est remplacée

Constructeurs

Une Collection/Map doit toujours avoir au moins deux constructeurs

- Un constructeur sans paramètre
  - new ArrayDeque(), new HashMap(), etc
- Un constructeur qui prend une autre collection/map en paramètre et recopie tous les éléments
  - ArrayDeque<E>(Collection<? extends E>)
  - ArrayList<E>(Collection<? extends E>)
  - HashMap<K,V>(Map<? extends K, ? extends V>)

Pré-dimensionnement

ArrayList, HashSet, HashMap, etc ont un redimensionnement (resize,rehash) qui peut coûter chère

On peut pré-dimensionner ces collections/maps

- ArrayList<E>(capacity)
- <E> HashSet.newHashSet(capacity)
- <E> LinkedHashSet.newLinkedHashSet(capacity)
- <K,V> HashMap.newHashMap(capacity)
- <K,V> LinkedHashMap.newLinkedHashMap(capacity)

Vue

Une implantation créée à partir d'une autre implantation peut ne pas stocker les données

```
var list = new ArrayList<>(List.of(1, 2, 3, 4));
var list2 = list.subList(2, 4);  // entre 2 et 4 exclu

list2.set(0, 42);
System.out.println(list);  // [1, 2, 42, 4]
```

ArrayList.subList() garde un pointeur sur les données

Vue non modifiable

Une vue non modifiable sur une implantation modifiable

ex: Collections.unmodifiableList()/unmodifiableSet() etc

```
class Library {
  private final ArrayList<Book> books;

  public Library(List<? extends Book> books) {
    this.books = new ArrayList<>(books);  // copie défensive
  }

  public List<Book> books() {
    return Collections.unmodifiableList(books);
  }
}
```

Nullabilité

Mantra depuis Java 1.2,

“Si on stocke pas null, on recoit pas null”

A partir de Java 5, les nouvelles collections ne permettent pas de stocker null

- ArrayDeque, ConcurrentHashMap, etc

Nullabilité et exception

Il y a 4 méthodes de Map qui considère que null veut dire “pas de couple clé/valeur”

- Si il n’y a pas de valeur associé à la clé
  - Map.**get**(clé) renvoie null (getOrDefault() est mieux)
  - Map.**compute**(clé, biFunction) appel la biFunction avec null en 2ième paramètre (merge() est mieux)
- Si la biFunction renvoie null, on supprime l’élément
  - Map.compute(clé, biFunction), computeIfPresent(clé, biFunction) et computeIfAbsent(clé, biFunction)

On ne renvoie pas null !

Une méthode qui renvoie une collection/map ne renvoie pas null mais une collection/map vide

```
public static Set<String> findBlahBlah() {
  if (…) {
    return null;  // ahhhh
  }
  ...
}
```

```
public static Set<String> findBlahBlah() {
  if (…) {
    return Set.of();  // ok !
  }
  ...
}
```

Ordre

L’ordre des éléments/clés dépend de l’interface ou de l’implantation

- ordre d’insertion (List, Deque, LinkedHashSet/Map)
- ordre d’acces (LinkedHashMap(capacity, factor, true))
- Trié avec fonction comparaison (TreeSet/Map)
- Sans ordre (Set)

Comparator<T>

Fonction de comparaison entre deux éléments de même type

```
interface Comparator<T> {
  int compare(T t1, T t2);
}
```

Même sémantique que strcmp

- <0 si t1 < t2
- >0 si t1 > t2
- ==0 si t1.equals(t2)

Attention aux overflow !

On ne peut pas utiliser '-' dans l'implantation

```
record Person(int id) {}

Comparator<Person> comparator =
   (p1, p2) -> p1.id() – p2.id();  // ahhh
```

Si id est Integer.MIN_VALUE ou proche, problème !

```
Comparator<Person> comparator =
   (p1, p2) -> Integer.compare(p1.id(), p2.id());  // ok
```

Ordre naturel

On peut définir un ordre naturel sur une classe avec l’interface Comparable

```
record Person(int id)
  implements Comparable<Person> {

  public int compareTo(Person p) {
    return Integer.compare(id, p.id);
  }
}
```

Attention, compareTo doit marcher avec equals !

Comparator.comparing()

Méthode statique qui créée un Comparator à partir d'une fonction de projection

- Comparator<Person> comparator =
  Comparator.comparing(Person::id); // ahh
- Comparator<Person> comparator =
  Comparator.**comparingInt**(Person::id); // ok

Le type de id doit être un Comparable !

Concurrence

Le package java.util.concurrent défini des implantations concurrent et lock-free

- List (CopyOnWriteArrayList)
- Set (CopyOnWriteArraySet, ConcurrentSkipListSet)
- Map(ConcurrentHashMap, ConcurrentSkipListMap)

Algos

- copy on write = on duplique le tableau à chaque mutation
- concurrent = on utilise l’instruction CAS et volatile
- skip list = liste trié (comparator) en O(ln n)

java.util.concurrent et size

Pour les collections

- de java.util
  - La complexité de la méthode size() est **O(1)**
- de java.util.concurrent
  - La complexité de la méthode size() peut être **O(n)**

La complexité de IsEmpty() est en **O(1)**

Parcours et Iterable

Les collections implante l’interface Iterable

```
interface Iterable<E> {
  Iterator<E> iterator();
}
```

La syntax for(:) permet d’utiliser un itérateur

```
for (var element : iterable) {
  ...
}
```

Le problème ?

On ne peut pas parcourir une collection avec un index

- car les Set et les Queue on pas de notion d'index
- car List.get(index) peut avoir une complexité de **O(n)**

Un itérateur (un curseur) permet de parcourir une collection en complexité **O(n)**

Problème de complexité

```
LinkedList<Integer> list = ...  // doublement chainée
for(var i = 0; I < list.size(); i++) {
  var element = list.get(i);  // ahhh
  ...
}
```

```
LinkedList<Integer> list = ...  // doublement chainée
for(var element: list) {    // ok
  ...
}
```

Iterator<T>

Curseur qui parcours les éléments

```
interface Iterator<T> {
  boolean hasNext();
  T next();
}
```

- **hasNext**(), y-a-t-il un suivant
- **next**() renvoie l'élément courant et passe au suivant

Parcours avec un itérateur

```
for (var element : iterable) {
  ..
}
```

est équivalent à

```
var iterator = iterable.iterator();
while (iterator.hasNext()) {
  var element = it.next();
  ...
}
```

Iterateur et Mutation

Que se passe-t-il si on modifie <u>structurellement</u> la collection lors du parcours ?

- Si collection non modifiable
  - UnsupportedOperationException lors de la mutation
- Si la collection est concurrente
  - On ne voit pas la mutation (*Snapshot At The Beginning*)
  - On peut voir la mutation (*Weakly Consistent*)
- Sinon on plante au prochain iterator.next() (*failfast*)
  - Lève ConcurrentModificationException

ConcurrentModificationException

Comme son nom ne l’indique pas, ce n’est pas un problème de concurrence

- Le problème est une mutation de la collection lors du parcours de cette collection

Résoudre le problème

- On enregistre les mutations et on les appliques à postériori
- On recréé une nouvelle collection avec un Stream
- On utilise les méthodes de l’itérateur

Opérations sur l’itérateur

Un itérateur possède des méthodes pour modifie la collection dont il est issue (pas de problème de *failfast*)

- Iterator.remove()
  Supprime l’élément renvoyé par next()
- ListIterator.add()
  Ajoute un élement après celui renvoyé par next()
- ListIterator.set()
  Change l’élement renvoyé par next()

ListIterator est un itérateur spécifique sur les List

Iterable et forEach()

Iterable définie aussi la méthode forEach
(par défaut)

- Iterable<E>.forEach(Consumer<? super E> consumer)

Le collections implantent

- L'itérateur (itération externe)
- Et la méthode forEach (itération interne)

La méthode forEach(consumer) est souvent plus efficace mais est limitée car les lambdas ne capture que les variables effectivement final

Iteration interne vs externe

```
record Boat(int price) { }
List<Boat> boats = …
```

Itération interne

```
var price = 0;
boats.forEach(boat -> price+ = boat.price(); );  // ahh
```

Itération externe

```
var price = 0;
for(var boat: boats) {
  price += boat.price();  // ok
}
```

Contrats
(partie 2)

java.util.Collection<E>

Interface de base des collections

- isEmpty/size()
- add*(E)
- contains(Object)/remove*(Object)
- equals(Object)/hashCode()/toString()
- toArray(intFunction) et toArray(T[])

Il n’y a pas d’ordre spécifié

* fait des mutations structurelles ou non

equals()

Est définie de façon étrange

- Il faut que les éléments soit égaux
- Pour une List, l’autre collection doit être une List
- Pour un Set, l’autre collection doit être un Set

```
List.of(1, 2, 3).equals(Set.of(1, 2, 3)) // false
```

Cela rend la définition non symétrique

```
collection.equals(list) != list.equals(collection)
```

Les 2 méthodes toArray()

T[] toArray(IntFunction<T[]> function)

- Prend une fonction de création de tableau en paramètre (par ex: String[]::new)
- Le type du tableau peut être différent du type de la collection (mais ArrayStoreException à l'exécution)

– T[] ToArray(T[] array)

- Si le tableau est plus petit, créé un tableau de la bonne taille
- Si le tableau est plus grand, ajoute un null après le dernier élément (ahhhh)

Exemple d’utilisation de toArray()

Avant Java 11,

```
List<Object> list = List.of("foo", "bar");
String[] array = list.toArray(new String[0]);
```

Après Java 11,

```
List<Object> list = List.of("foo", "bar");
String[] array = list.toArray(String[]::new);
```

Le fait que chaque élément est une String est testé à l’insertion dans le tableau

java.util.List

- Impose l’ordre d’insertion
- Méthodes supplémentaires
  - E get(index), set*(index, E), add*(index, E), E remove*(index)
  - int indexOf(E), int lastIndexOf(E)
  - replaceAll*(unaryOp)
  - List<E> subList(start, end)
  - sort*(comparator)

Méthodes indexées et contains()

Méthodes rarement utilisées en pratique

- Les méthodes get(index) et set(index, E) peuvent être en **O**(n)
- Les méthodes add(index, E) et remove(index) sont en **O**(n)

Rechercher dans une List est lent !

- contains(Object) est aussi en **O**(n)

Premier et dernier élément

Il n’existe pas de méthode pour obtenir le premier ou le dernier élement

premier élément

```
var first = list.get(0);  // O(1)
```

dernier élément

```
var last = list.get(list.size() - 1);  // O(1)
```

Une implantation de List doit garantir que l’on accède au premier ou au dernier élément en temps constant

Relation avec les tableaux

Normalement, on devrait utiliser un tableau et pas une List si l'on connait le nombre d'éléments à l'avance

Mais comme créer un tableau de type paramétré est impossible

- On utilise des List même si on connait la taille à l'avance

java.util.Set

Ensemble sans doublons

Ne possède pas de méthodes supplémentaires par rapport à l’interface Collection

Permet la recherche et la suppression soit en O(1) soit en O(ln n)

java.util.Deque

Pile, file, etc par défaut ajoute à la fin et retire au début

Méthodes supplémentaires

- offer(E) / offerFirst(E) / **offerLast**(E)
  ajoute si possible
- E poll() / E **pollFirst**() / E pollLast()
  retire si possible, sinon renvoie null
- E peek() / E **getFirst**() / E getLast()
  valeur sans suppression, ou null si pas de valeur

Relation avec les collections

Les méthodes add(), remove() est l’équivalent de offer(), poll() mais lève une exception si l’opération n’est pas possible

La méthode element() lève une exception si il n’y a pas de premier élément

java.util.Map

Dictionnaire qui associe à une clé une valeur,
les clés n’ont pas de doublons

Méthodes

- V get(clé), V getOrDefault(clé, V defaultValue),
  V put(K, V), V putIfAbsent(K, V)
- replace(K, V), replace(K, V, V), remove(Object), remove(Object, Object)
- computeIfAbsent(K, Function<K,V>)
- merge(K,V, BiFunction<V,V,V>)
- Set<Map.Entry<K, V>> entrySet(),
  Set<K> keySet(), Collection<V> values()
- forEach(BiConsumer<K,V>)

Map.computeIfAbsent(clé, function)

Permet de ne pas recalculer une valeur déjà calculée (cache)

```
class Cache {
  static int myFunction(int value) {
    ...
  }

  private final HashMap<Integer, Integer> map =
    new HashMap<>();

  public int lookup(int value) {
    return map.computeIfAbsent(value, v -> myFunction(v));
  }
}
```

Map.merge(K,V, BiFunction<V,V, V>)

Calculer le nom de fois qu’apparait un mot dans une liste

```
public static Map<String, Integer> group(List<String> list) {
  var map = new HashMap<String, Integer>();
  for(var word: list) {
    map.merge(word, 1, (v1, v2) -> v1 + v2);
  }
  return map;
}
```

Map.Entry<K,V>

Interface réprésentant une pair (un couple) de clé / valeur

```
interface Map {
  interface Entry<K,V> {  // à l'intérieur de l'interface Map
    K getKey();
    V getValue();
    default void setValue(V value) { throw new UOE(); }
  }
…
}
```

entrySet()

Il n’existe pas d’itérateur sur une Map

La méthode entrySet() est une vue des couples clé / valeur de la Map

```
for (var entry : map.entrySet()) {
  var key = entry.getKey();
  var value = entry.getValue();
  ...
}
```

Toutes modifications de la Map est reflètées dans la vue renvoyée par entrySet()

keySet() et values()

Comme entrySet(), keySet() et values() sont des vues de la Map

- Ensemble des clés

```
for (var key : map.keySet()) {
  ...
}
```

- Collection des valeurs

```
for (var value : map.values()) {
  ...
}
```

Map.forEach(BiConsumer<K,V>)

Equivalent à un parcours de l’entrySet() mais plus efficace

- Par contre, on est limité par la capture des lambdas

Parcours les clés/valeurs

```
map.forEach((key, value) -> {
  ...
})
```

Implantation des collections (partie 3)

Implantations des List

Implantations les plus importantes

- ArrayList<E>: tableau dynamique
- Arrays.asList(E..): vue d’un tableau
- List.of(E…): implantation non modifiable
- Collections.nCopies(int n, E value): n fois le même élément
- CopyOnWriteArrayList<E> implantation concurrente

Implantations des Map

Implantations les plus importantes

- HashMap<K,V>: table de hachage
- LinkedHashMap<K,V>: maintien ordre d’insertion
- TreeMap<K,V>: arbre rouge noir, triée
- Map.of(K,V, ...): implantation non modifiable
- EnumMap<K extends Enum<K>, V>: les clés viennent d’un même enum
- IdentityHashMap<K,V>: utilise == et System.identityHashCode() au lieu de equals et hashCode
- ConcurrentHashMap<K,V>: version concurrente
- ConcurrentSkipListMap<K,V>: version concurrente triée

Implantations des Set

Implantations les plus importantes

- HashSet<E>: table de hachage
- LinkedHashSet<E>: maintien ordre d’insertion
- TreeSet<E>: arbre rouge noir, triée
- Set.of(E...): implantation non modifiable
- EnumSet<E extends Enum<E>>: les éléments viennent d’un même enum
- CopyOnWriteArraySet<E>: version concurrente
- ConcurrentSkipListSet<E>: version concurrente triée

Implantation des Deque

Implantations les plus importantes

- ArrayDeque<E>: tableau circulaire
- ConcurrentLinkedQueue: version concurrente
- ArrayBlockingQueue: version conurrente et bloquante

Ecrire sa propre implantation

Il existe des classes abstraites qui implantent la plupart des méthodes à partir d’autres

Offre uniquement une version non modifiable

- AbstractList pour les List
- AbstractSet pour les Set
- AbstractMap pour les Map
- AbstractQueue pour les Deque

Implanter une List

La classe abstraite AbstractList fourni une implantation pour les listes non modifiables à accès constant (get(index) en temps constant)

Méthodes à implanter

- int size()
- E get(index)

Il faut aussi implanter l'interface RandomAccess pour indiquer que l'implantation de get() est en O(1)

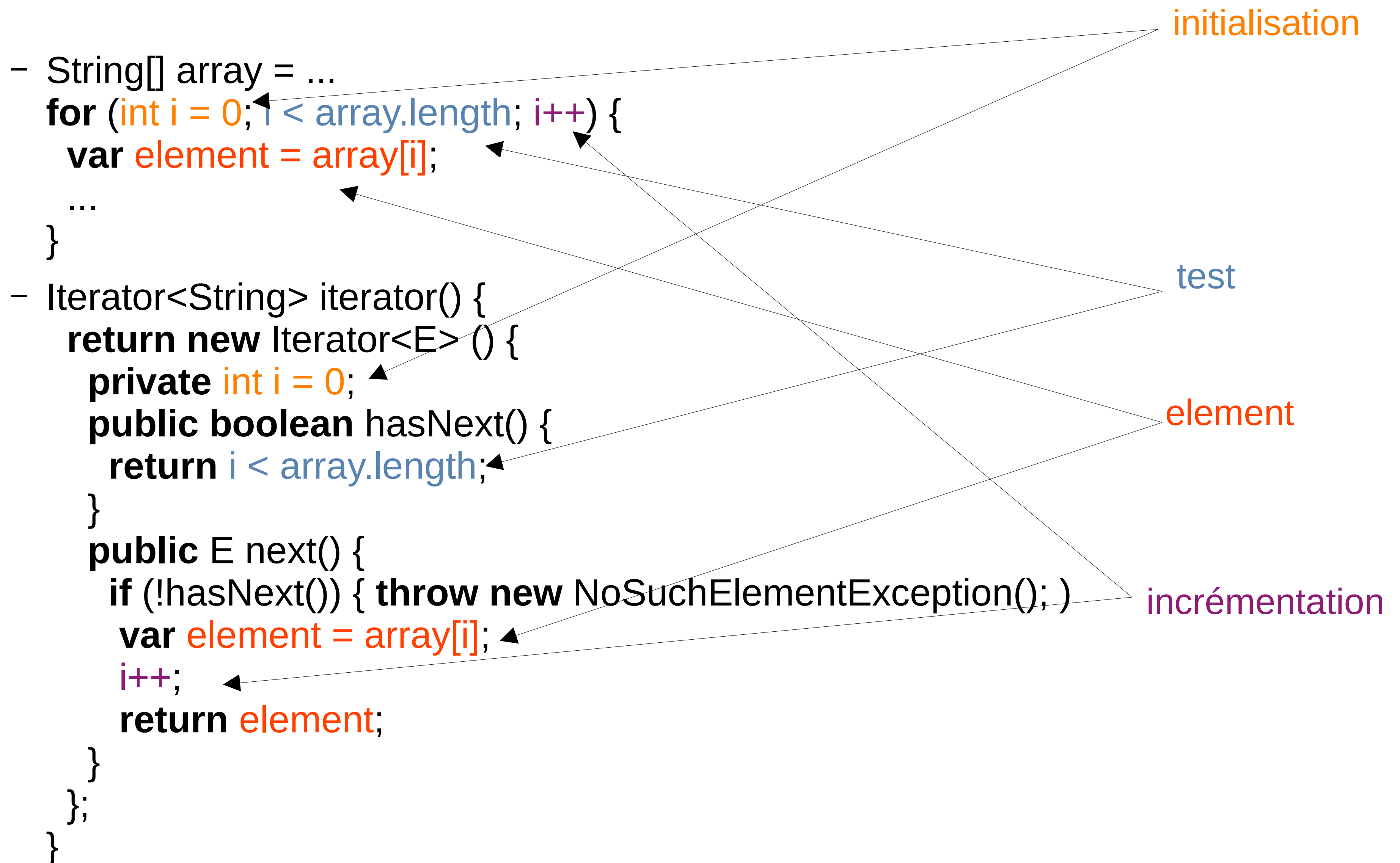
Iterator<E>

Ecrire un Iterator demande d’écrire au moins les méthodes

- boolean hasNext()
  y a-t-il un élément suivant
- E next()
  renvoie l’élement courant et passe au suivant

Attention, next() doit lever NoSuchElementException si il n’y a pas d’élément courant

Comment écrire son Itérateur



Implanter un Set

La classe abstraite AbstractSet fourni une implantation pour les ensembles non modifiables

Méthodes à implanter

- int size()
- Iterator<E> iterator(index)

mais contains() est en O(n), donc il faut la redéfinir

- boolean contains(Object)

Implanter une Map

La classe abstraite AbstractMap fourni une implantation pour les dictionnaires non modifiables

Méthodes à implanter

- int size()
- Set<Map.Entry<K,V>> entrySet()

mais get/getOrDefault()/containsKey() sont O(n), donc il faut les redéfinir

- V get(Object)
- V getOrDefault(Object, V defaultValue)
- boolean containsKey(Object)

Implanter une Queue

La classe abstraite AbstractQueue fourni une implantation pour les files non modifiables

Méthodes à implanter

- boolean offer(E)
- E poll()
- E peek()

Classe *Legacy* (partie 4)

Interfaces/Classes démodées

Il existe un certains nombres d’interfaces/classes qui

- Soit n’ont jamais vraiment été utilisées
- Soit ont été utilisée mais remplacées
  - À cause de problème algorithmique et/ou de performance

Classes Avant Java 1.2

Les classes avant l’API des collections ont leurs méthodes synchronized sauf l’itérateur

- Lent si on a pas besoin de concurrence
- Lent par rapport aux classes de java.util.concurrent
- Dangereuses quand on utilise un itérateur

Classes et leur remplacement

- Vector est remplacée par ArrayList
- Stack est remplacée par ArrayDeque
(il y a les méthodes push/pop/isEmpty)
- Hashtable est remplacée par HashMap
- Enumeration est remplacée par Iterator

Classe plus assez efficace

LinkedList

- Liste doublement chainées, trop lente par rapport aux implantations à base de tableau
- A remplacer par
  - ArrayList ou ArrayDeque (insertion au début)

AbstractSequentialList

- La classe abstraite implantée seulement par LinkedList

Classe avec un problème d’algo

PriorityQueue

- Permet de trier les éléments avec un tas
- Mais l’algo est pas *stable*
  - Deux objets égaux au sens de equals peuevnt être insérer dans un ordre et récupérer dans l’autre

Interfaces peu utilisées

OrderedSet, NavigableSet, OrderedMap, NavigableMap sont des interfaces pour TreeSet et TreeMap

Il est rare d’avoir des API qui prennent ces interfaces en paramètre

Collections non modifiables

Collections.emptyList()/emptySet()/emptyMap() et Collections.singletonList()/singleton()/singletonMap()

Remplacées par List.of(), Set.of() et Map.of()

- Qui marche de concert avec List/Set/Map.copyOf()

Implantations de Map.Entry

Obscures implantations jamais vraiment utilisées

- AbstractMap.SimpleImmutableEntry<K,V>, AbstractMap.SimpleEntry<K,V>

Remplacé pour la version non modifiable par Map.entry(K, V)
(attention c’est une méthode !)